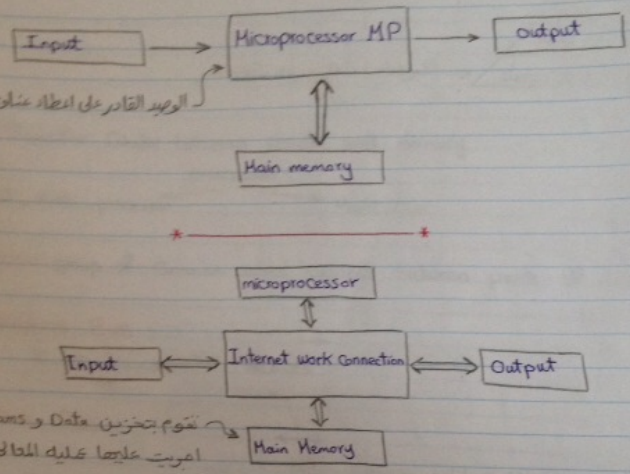
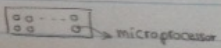
Ch 1 Microprocessor (Introduction)



Control processing Unit (cpu) ≡ Machine



→ cpu components

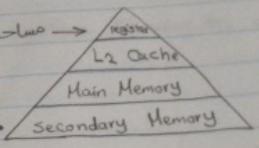
1. ALU
2. CU (Control Unit)
3. Registers level 1 cache

تتكلم في وظائف الجهاز

→ cpu functions:

1- Execode arithmatic & logic instruction
(Fetch - decode - execode)

→ Manage the program flow (make decisions)
* Zero * negative * overflow ...



→ Transfer Data between itself and memory

processor

الشغال → Multi-processor ← Multi-Core

Buses: group of Common wires connect between parts of device.

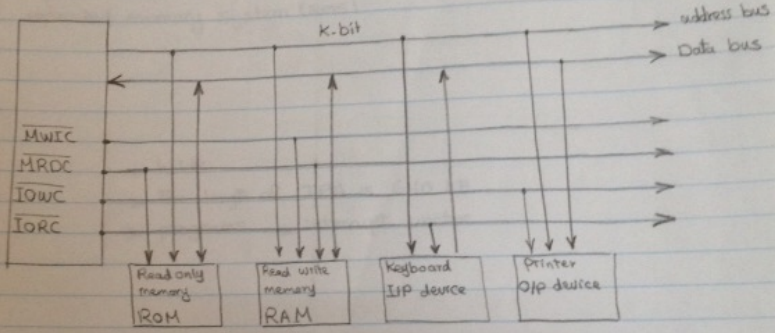
1. **Address Bus:** →
   K-bit ⟶ 2^K devices.

2. **Data Bus:** ↔
   MP
   ← كلما زاد كلما زاد معدل نقل البيانات من وإلى الـ MP
   devices ↔ MP

3. **Control Bus:** ↔
   ← ينقل Control Signal
   devices ↔ MP
   كلما زاد كلما كان أفضل

←

Draw the block diagram of a computer System showing the address, data and Control bus signal ?



* Simple Control Signal

MWIc	memory write Control Signal
MRDC	memory read Control Signal
IOWC	I/O write Control Signal
IORC	I/O read Control Signal.

→ MP → device التعامل بين

1- MP ⇝ send address

2- Control Signal MRDC

3- Data transfer.

The Memory System is divided into "3 main points"

1- Transient program area (TPH)
2- System Area
3- extended memory System (XMS)

1

→ holds
→ The length of TPA is 640 KB
→ programs → Interrupt vector

Ch 2 : The Microprocessor & Its Architecture

8 bit	16 bit	8 bit	
AH	AX	AL	Accumulator
BH	BX	BL	Base Index
CH	CX	CL	Count
DH	DX	DL	Data
	SP		Stack Pointer
	BP		Base Pointer
	DI		Destination Index
	SI		source index

IP
FLAG

CS	Code Segement
DS	Data Segement
ES	Extra Segement
SS	Stack Segement
FG	
GS	

special purpose.

Programming Model Of The Intel 8086 Pentium 4

32 bit	8bit	16bit	8bit	
EAX	AH	AX	AL	Accumulator
EBX	BH	BX	BL	Base Index
ECX	CH	CX	CL	Count
EDX	DH	DX	DL	Data
ESP		SP		Stack Pointer
EBP		BP		Base Pointer
EDI		DI		Destination Index
ESI		SI		Source Index

	IP	Instruction Pointer
	FLAGS	Flags

CS	Code
DS	Data
ES	Extra
SS	Stack
FS	
GS	

The Programming Model Of The Intel 80386 Pentium 4

كل 4 bit في الشاشة يعطي 1 في السادس عشر ←

20 bit → (XXXXX) Hexa.

Real Mode

Segement	Off Set	Special Purpose
CS	IP - EIP	Instruction address
SS	SP or BP, ESP / EBP	Stack address
DS	BX . DI . SI 8 bit 16 bit 3 numbers	Data address
ES	DI , EDI For string Instructions	String destination address
FS _ GS	No default	General address.

Programming Model

program Visible
- They are used during app-prog
- Ex: EAX, AX, AI, EBX ...
→ Mov AX, BX

program invisible
- not addressed directly during app-prog
- only 80286 - P4 Contain this type
- ex: FIAG, EFIAG

SUB AX, BX بعد تنفيذ أمر

Purpose Of Use

General Purpose
EAX , EBX
Hold Data address or off set +
البعد عن البدايه

Special purpose.
CS, DS, ES, SS

Int X = 3;
address ↑ data ↑

Addressing modes (How the memory is accessed)

Addressing mode ← كتيره لكتره طرق تنظيم البيانات داخل الذاكره
وظيفتها الوصول إلى الذاكره

1- Real mode 1 MB only
2- Protected mode upto 4GB

« Real mode »

→ Accumulator EAX . AX . AL . AH
Can be used as 32bit → EAX , 16 → AX , 8bit → AH, AL

II Special purpose :
← طرق أساسى فى عمليات الضرب والقسمة

→ Div AX , BX : $\frac{AX}{BX}$

AX : يوضع به الجزء الصحيح
DX : (Data register) يوضع به باقى القسمه "الكسر"

→ Mux AX^{16} , BX^{16}
16 * 16 → 32 bit

→ Mux EAX , EBX
32 , 32

DX	AX
high bytes	low bytes

EDX	EAX
high bytes	low bytes

Base Index

Can be used as 32 bit → EBX , 16 bit → BX
8 bit → BH , BL

General purpose → hold off set For Data Segement

Note : In Real Mode

We have 64 KB processor , 5 bit memory

→ To access the memory الوصول للذاكرة

← processor الـ من

$64K = 2^6 \cdot 2^{10} = 12^{16}$

$(16\ bit)_8 \rightarrow (4\ bit)_{Hexa.}$

~ From 4 bit processor how to reach 5 bit memory
we add zero from the left side to Base

→ to reach to any address in the memory we need Base and off set
الـ Base is in Code Segement

∴ Current address$_{(real)}$ = $(CS)_0$ + off set

Ex: CS = 4000 Hex , IP = 2342

Solution

Current address = $(CS)_0$ + IP = 40000 + 2342 = 42342 Hex

Count

To Count in any process with loop we need big register

loop CX , ECX

Repetition CX

Shift CL (we need small number of loops

Data

Can be used as 32 bit → EDX 16 bit → DX
8 bit → DH, DL
General purpose → Div, Mux الضرب والقسمة

Stack pointer

Can be used as 16 bit → SP
hold the off set of stack segement (SS) ← الأولوية له

Base pointer

Can be used as 16 bit → BP
used as off set for stack segement (SS)

Destination Index

Can be used as 16 bit → DI له الأولوية بعد BX
hold the off set of Data segement

Source Index

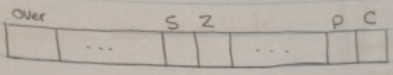
Can be used as 16 bit → SI له الأولوية بعد DI
hold the off set of Data segement

Instruction Pointer

Can be used as 16 bit → IP
hold the off set of current instruct to be excuted (in Code segement)

Ch 4 : Data Movement Instructions

Flag . Invisible



Can be used as 16 bit - 32 bit

→ Help microprocessor to take the correct way

[1] over Flow happened when → +ve + +ve = -ve
→ -ve + -ve = +ve

[2] Carry (C)

11111010

Carry 11101110 +
① ← 11101000

[3] Parity (P)

1 → number of one's is even

0 → number of one's is odd

تستفيد من هذا عندما يتم نقل Data من مكان لآخر نتأكد من أن Data تم نقلها بدون فقد صيت اذا كان عدد 1 زوجي بعد النقل وكذلك P=1 فان النقل تم بالصوره المطلوبه والعكس يكون فيه فقد في Data

Ex: SS = 3245 H

BP = 2342

Sol

Base address = 33450 H

Current address = 33450 + 2342 = 35792

Ending address = Starting + FFFF

= 33450 + FFFF

Used in two dimension ....

Ch3 : Addressing modes (Real Mode)

1] Immediate Addressing Mode

I → R (I: information, R: register)

Examples:
- Mov BL, 44
- Mov BH, 44H
- Mov CL, 1001110 B
- Mov ECX, FC23F34EH

2] Register Addressing Mode

R ↔ R (register ↔ register)

- Mov CL, BL (✓)
- Mov CX, BX (✓)
- Mov EAX, EBX (✓)
- Mov DS, CX (✓)
- Mov DS, CS (✗)
- Mov CS, BX (✗)
- Mov BX, CS (✓)

3] Direct Addressing Mode

- Direct addressing
  - R: EAX, AX, AH, AL
- Displacement
  - (offset) addressing
  - Any Register.

R ↔ M (memory)

Direct	Displacement
Mov AL, Num	Mov CL, Num
Mov AX, Num	Mov CX, Num
Mov Num, AX	Mov EBX, Num
Mov EAX, Num	Mov DX, [1023]

4] Register Indirect Addressing Mode[]

(BP, BX, SI, DI)

R ↔ [M]

- Mov AX, [BX]
- Mov [BP], DL
- Mov EDX, [DI]
- Mov [BX], [SP] (X)

5] Bass_Plus_Index Addressable Mode

EBX → BX, EBP → BP

ESI → SI, EDI → DI

use with one dimention array

R ↔ M

- Mov DX, [BX+DI]
- Mov WORD / DWORD PTR [BX+DI], DX

6] Register_Relative Addressable_Mode

BP, BX, SI, DI

EBP, EBX, ESI, EDI

off set

مكان في الذاكرة

R ↔ M

- Mov AX, [BX+1000]
- Mov AX, 1000 [DI]
- Mov WORD / DWORD PTR [BX+1000], AX

7] Base_relative plus Index Addressable Mode

Used in two dimention Array

R ↔ M

- Mov WORD PTR 1000 [BX+DI], A
- Mov DH, [BX+DI+20H]

Effective Address (EA)

No EA

Notes

( )10 عشرى

( )B Binary ثنائى

( )H السادس عشر ، Characters

R ↔ R has the same size 8 ↔ 8, 16 ↔ 16

لا يجوز بأى حال من الأحوال تغيير Code segment (CS) الذى يحتوى على عنوان الأوامر داخل الذاكرة وكذلك لا يجوز النقل منه segement الى آخر

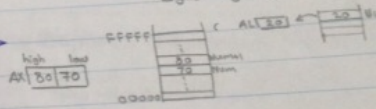
R source (R_s)

Register الذى نحصل منه على البيانات

E.A = memory = Num

1→ Memory ~ Byte addressable

Byte by Byte ترقيمه



2→

little endian assignement

$EA = (seg)_{+0} + off\ set$

① $EA = (DS)_0 + BX$

② $EA = (SS)_0 + BP$ ③ $EA = (DS)_0 + DI$

عند نقل بيانات من Register الى Memory لابد من تحديد المساحه المطلوب تخزين البيانات بها حيث أن Memory byte addressable

Mov [BX], AL ← لم يتم تحديد مساحه التخزين لذا يتم التعديل الى

Mov Byte PTR [BX], AL

Mov WORD PTR [BX], AL

Mov DWORD PTR [BX], AL

$EA = (DS)_0 + BX + DI$

Starting address

$EA = (DS)_0 + BX + 1000$

ويظل effective address ثابت

عند نقل بيانات من Memory الى Register لا تحدد المساحه للتخزين ولا توجد مشاكل

$EA = (DS)_0 + \begin{matrix} BX \\ DI \\ 1000 \end{matrix}$

Ch 4 : Data Movement Instructions

1- Assembler & Deassembler

2- Stack (push, pop)

3- Instructions

Assembler & Deassembler

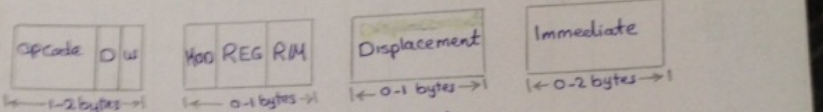
Hint: Assembler: transform the assembly code to machine code

Deassembler: transform the machine code to assembly code

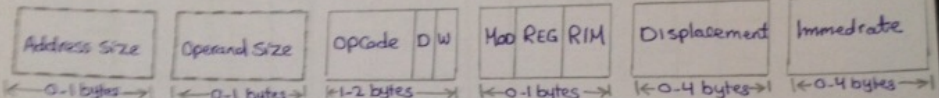
→ machine code (0's, 1's)

→ Modes of Instruction

1. 16 bit instruction mode (8086)



2. 32-bit Instruction mode (80386 above)



To Transfere from assembler to deassembler Follow this steps

1

Operand Size	Address Size	mode
66 H X	67 H X	- 16-bit instruction mode with its default (8 - 16 bit) registers w=0 w=1
✓	X	- 16 bit instruction but works with (8 - 32 bit) registers. w=0 w=1
X	✓	- 32 bit instruction works with its default (8 - 32 bit) register w=0 w=1
✓	✓	- 32 bit instruction but works with (8-16 bit w=0 w=1 registers.

2 | Opcode | D | W |

→ Opcode: has machine code of the instruction (Mov, sto, swap)
we get this code from the table
like Mov ⟶ 100010

→ D ⟶ D=1 (when we transfer from Memory to register)
⟶ D=0 (when we transfer from register to Memory)

→ W ⟶ W=1 (word / Dword)
⟶ W=0 (Byte)

3 | MOD | REG | R/M |

→ MOD : Its machine code can be get from this table ↷

MOD	Function		Ex:
00	No Displacement	R/M is memory	Mov AL, [DI]
01	8-bit sign-extended displacement		Mov AL, [DI+2]
10	16-bit displacement		Mov AL, [DI + 1000₁₀]
11	R/M is a register		

→ REG : Its machine code can be get from this table

Code	W=0 (Byte)	W=1 (word)	W=1 (Dword)
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

Code	Seg
000	ES
001	CS
010	SS
011	DS
100	FS
101	GS

→ R/M ⟶ to get the Code of R/M → at First we want to know if R/M is Memory or Register that depend on MOD

IF MOD = 11 so R/M is registers
we can get its Code from register table.

Code	W=0 (Byte)	W=1 (word)	W=1 (Dword)
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

IF MOD = (00, 01, 10) so R/M is Memory
we Can get its Code
From this table

R/M Code	Addressing mode	Function
000	DS : [BX + SI]	DS : [EAX]
001	DS : [BX + DI]	DS : [ECX]
010	SS : [BP + SI]	DS : [EDX]
011	SS : [BP + DI]	DS : [EBX]
100	DS : [SI]	Uses scaled-index byte
101	DS : [DI]	SS : [EBP]*
110	SS : [BP]* special addressing mode	DS : [ESI]
111	DS : [BP]	DS : [EDI]
	16-bit R/M memory addressing mode	32-bit addressing mode selected by R/M

4 Displacement

① 1 byte □□ ⟶ Put as it توضع كما هي
ex: 1 → 01 , 21 → 21

② 2 byte □□ □□ ⟶ Swap high, low

ex : 1234 → 3412

ex : 123 → 2301

Stack
(pop & push)

→ The PUSH and POP instructions are important instructions that store and retrieve data from the LIFO (last In First Out) Stack memory.

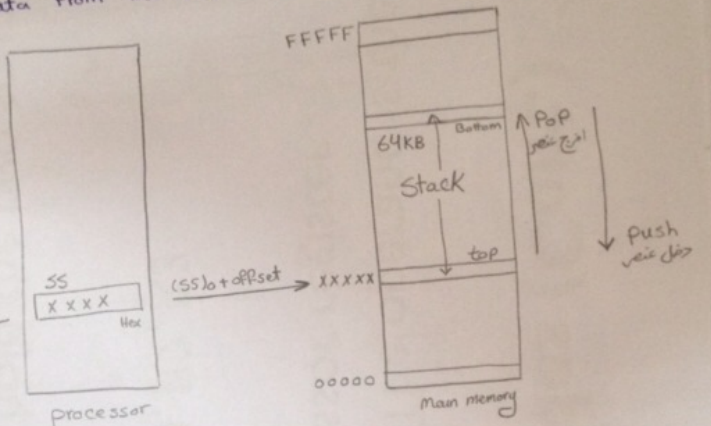
Stack (LIFO)

↓

two operations

Pop — يخرج عنصر ويرجع Pointer

Push — Pointer يرجع ويدخل عنصر



عندما Stack بالعناصر ← ووضع عنصر آخر بها يحدث Cyclic nature

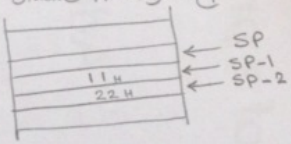
اي يتم دفع ما بالعنوان الاول ووضع العنصر الجديد في مكانه

→ Push AX (✓)

AX | 11H | 22H |

high byte — low byte

Stack في AX محتوى وضع



→ Push 5262H (✓)

→ Push [BX] (X) ← هنا لم يحدد المساحة التي يخزن بها البيانات في memory

EA = (DS)o + BX

The Correct

Push Byte PTR [BX] (✓)

Push (Word - DWord) PTR [BX] (✓)

→ Push CS (✓) → هنا لم يغير محتوى CS

---

* Hinit

22H Can be Put in memory as?

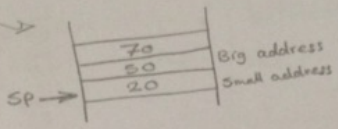
1. Byte | 22H |
2. word | 00 | 22 |
3. DWord | 00 | 00 | 22 |

Pop ← stack or data

Pop 2345 H (X)

Pop CS (X) → we mustn't change CS

Pop BX (✓) ↝



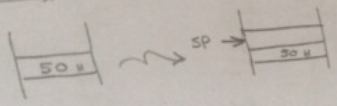
① BX | 50 | 20 |
high low

high address → high byte
low address → low byte

② Then SP = SP + 2

Pop Byte PTR [BX]

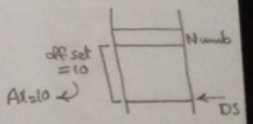
EA = (DS)$_0$ + BX →



Instructions

- LEA
  - LDS, LSS, LFS, LGS, LES
  - * off set
- LODS, STOS, MOVS, INS, OUTS.
  - * IN, Out
  - * X ChG
  - * BSwap
  - * CMOS

→ I LEA (load effective address)



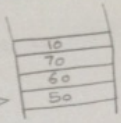
ex: LEA AX, Numb ≡ Mov Ax, off set Numb

هذا الأمر معناه وضع بعد المكان Numb عن البداية في AX

ex: LEA BX, [DI] ↝ EA = (DS)$_0$ + DI ≠ MOV BX, off set [DI] (X)

→ LDS

ex: LDS BX, [DI]



EA = (DS)$_0$ + DI = XXXXX →

BX	60	50
DS	10	70

← يتم تحديد برايه العنوان من EA

← تأخذ من memory بيانات تحجها

حجم المكان اللى سوف توضع به

Hinit

bit = 2 byte

فى المثال BX تحجز 16 bit اى

∴ تأخذ مكانين من الذاكره 4 byte

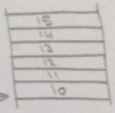
وتضعهم فى BX بحيث يوضع العنوان

الأول من الذاكره فى byte low والمكان الثانى فى high byte

← ثم تأخذ العنوان التالى وتضعهم فى DS حيث

آن الأمر LDS

→ LSS



ex: lss EAX, [BX+DI]

EA = (DS)$_0$ + BX + DI = XXXXX →

EAX	13	12	11	10

where EAX → 32 bit

SS	15	14

→ LES

ex: LES BX, [DI]

EA = (DS)$_0$ + DI = XXXXX →

BX	60	50

, ES

ES	80	70

→ LFS

ex: LFS BX, [DI]

EA = (DS)$_0$ + DI = XXXXX →

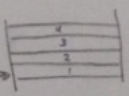
BX	60	80

, FS

FS	70	50

→ LGS

ex: LGS BX, [DI]



EA = (DS)$_0$ + DI = XXXXX →

BX	2	1

, GS

GS	4	3

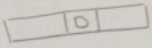
LCS EAX, [DI] (X)

because CS mustn't be Changed.

String Instructions

LODS, STOS, MOVS, INS, OUTS

This Instructions depend on direction "D"

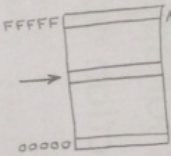


D=0 ⇝ Automcrement mode

تسير فى اتجاه زياده العنوان

D=1 ⇝ Autodecrement mode

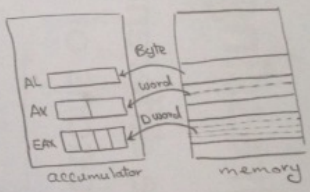
تسير فى اتجاه نقص العنوان



II LODS

تأخذ من memory وتضع فى accumulator

ثم أزود أو أقلل حسب D



LODSB → byte

LODSW → word

LODSD → double word

المكان غير معروف مباشره يوصله EA عن طريق

LODS list → byte (DB)

LODS Outer → word (DW)

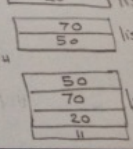
LODS FROG → double word (DD)

المكان الذى تريد وضع فيه فى الذاكره معروف مباشرة

list: DB 20 H

list2: DW 5070 H

list3: DD 5070 2011 H



From Source → to Destination

Data Segment — offset SI

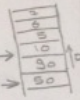
Extra Segment — DI

ex: DS=1000H, SI=1000H, D=0
execute LODSW then LODSD

Solution

EA = (DS)0 + SI = 10000 + 1000 = 11000

SI new = SI old + 2bit = 1002



AX | 90 | 50 |

To execute LODSD

SI = 1002, DS = 1000, D=0

EA = 10000 + 1002 = 11002

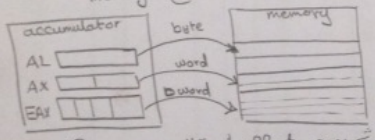
EAX | 7 | 6 | 5 | 10 |

SI new = SI old + 4bit = 1002 + 4 = 1006

If D=1 SI new = SI old − (1 → byte, 2 → word, 4 → Dword)

2 STOS

يأخذ من accumulator ويضع في memory



ثم يزود أو يقلل offset حسب D

STOSB, STOSW, STOSD
STOS list (DB), STOS list2 (DW)
STOS list2 (DD)

3 MOVZX

زود باقي خانات Register بـ اصفار Zeros

MOVSX (Memory) زود باقي المساحه بنفس القيمه حسب الاشاره

4 XCHG بدل الاماكن

→ Both places must have the same size
→ CS mustn't be Change

Examples:

XCHG AL, CL (✓)
XCHG CX, BP (✓)
XCHG Data, AX (✓)
XCHG DS, ES (X)
XCHG CS, AX (X)

5 BSWAP

→ work only with 32 bit registers (80486 → P3, P4)

EAX | 50 | 20 | 30 | 40 | before

EAX | 40 | 30 | 20 | 50 | after

6 CMOV (Conditional Mov) نقل مشروط

CMOVZ → هذا الأمر معناه
1- يرى خانه Z في Flag
Z=0 لايفعل شيء
Z=1 يفعل

ex: CMOVZ AX, BX
IF Z=1 ∴ AX = BX
else Do nothing

→ CMOVC → Chick Carry
IF C=1 → Move
else Do nothing

→ CMOVNC → Chick No Carry
IF C=0 → Move
else Do nothing